

Department I - C Plus Plus

Modern and Lucid C++
for Professional Programmers

Week 12 – Dynamic Polymorphism

Thomas Corbat / Felix Morgner
Rapperswil, 1.12.2020
HS2020



INSTITUTE FOR
SOFTWARE

- You can safely employ virtual dispatch
- You can explain the dangers when working with value semantics and dynamic dispatch
- You can use virtual base classes with `std::unique_ptr`

- **Recap Week 11**
- **Dynamic Polymorphism**
 - Motivation for Inheritance
 - Review of Inheritance
 - Shadowing vs. Virtual Member Functions
 - Possible Problems
 - Guidelines

Recap Week 11

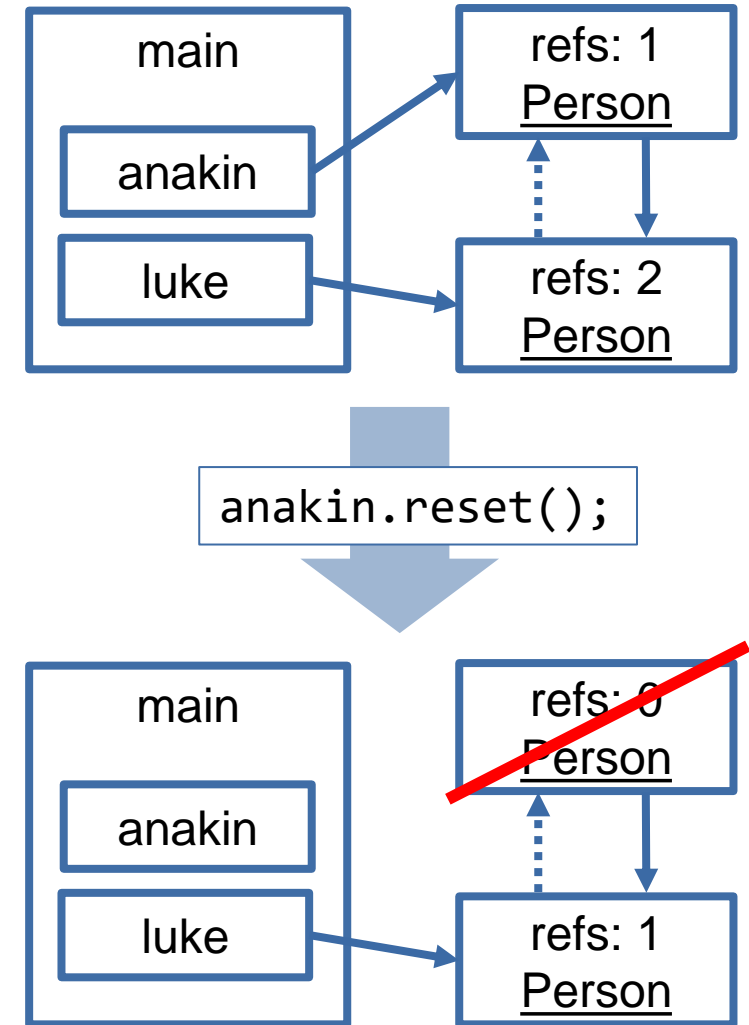


```
std::unique_ptr<X> factory(int i) {  
    return std::make_unique<X>(i);  
}
```

- **std::unique_ptr<T> obtained with std::make_unique<T>()**
- **std::shared_ptr<T> obtained with std::make_shared<T>()**
- **std::make_unique<T>() and std::make_shared<T>() are factory functions**
- **With these smart pointers you don't have to call delete ptr; yourself**
- **Still: Always prefer storing a value locally as value-type variable (stack-based or member)**

- The `std::shared_ptr` cycles need to be broken

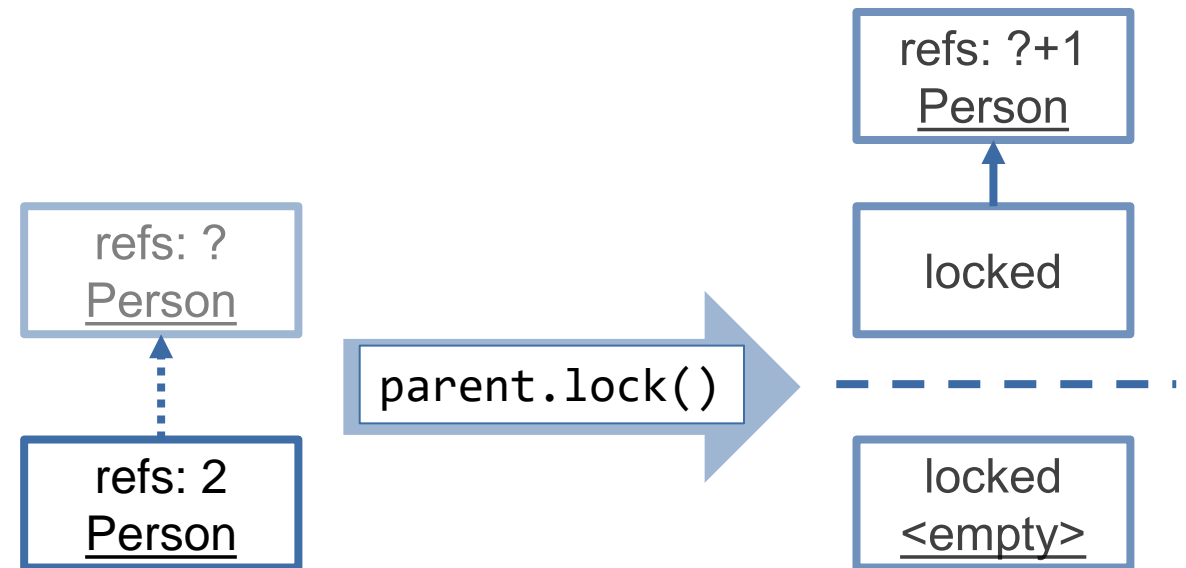
```
struct Person {  
    std::shared_ptr<Person> child;  
    std::weak_ptr<Person> parent;  
};  
  
int main() {  
    auto anakin = std::make_shared<Person>();  
    auto luke = std::make_shared<Person>();  
    anakin->child = luke;  
    luke->parent = anakin;  
    //...  
}
```



- A `std::weak_ptr` does not know whether the pointee is still alive

- `std::weak_ptr::lock()` returns a `std::shared_ptr` that either points to the alive pointee or is empty

```
struct Person {  
    std::shared_ptr<Person> child;  
    std::weak_ptr<Person> parent;  
  
    void Person::acquireMoney() const {  
        auto locked = parent.lock();  
        if (locked) {  
            begForMoney(*locked);  
        } else {  
            goToTheBank();  
        }  
    }  
};
```



Dynamic Polymorphism



- **Mix-in of functionality from empty base class**

- Often with own class as template argument (CRTP) e.g., `boost::equality_comparable<T>`
- No inherited data members, only added functionality

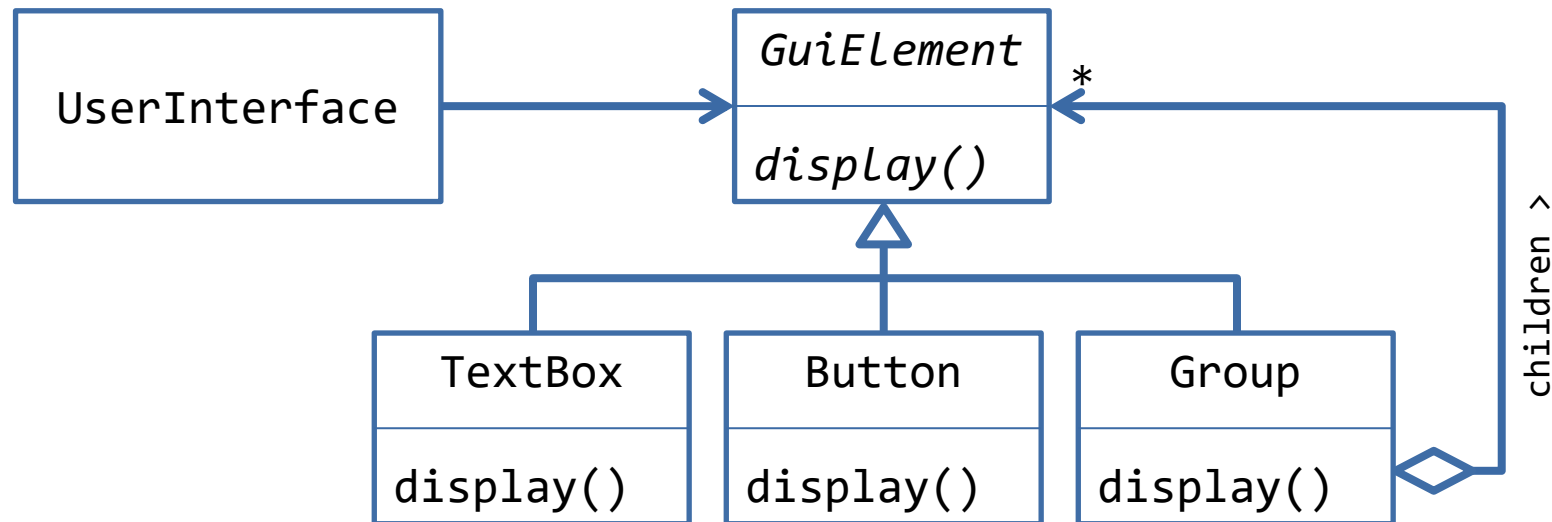
```
struct Date : boost::equality_comparable<Date> {  
    //...  
};
```

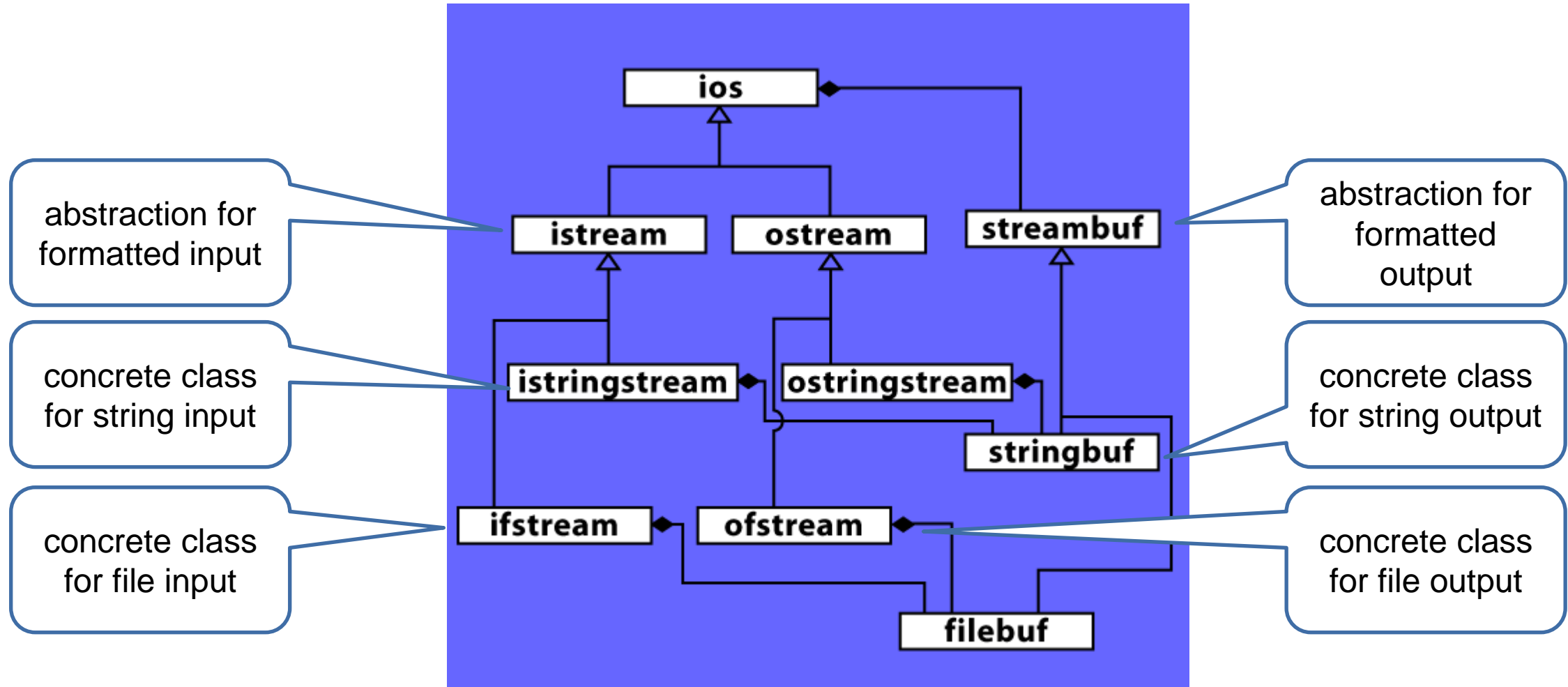
- **Adapting concrete classes**

- No additional own data members
- Convenient for inheriting member functions and constructors

```
template<typename T, typename Compare>  
struct indexableSet : std::set<T, Compare> {  
    //...  
};
```

- **Implementing a design pattern with dynamic dispatch**
 - e.g., Strategy, Template Method, Composite, Decorator
 - Provide common interface for a variety of dynamically changing or different implementations
 - Exchange functionality at run-time
- **Base class/interface class provides a common abstraction that is used by clients**





```
class Base {};  
class DerivedPrivateBase : Base {};  
struct DerivedPublicBase : Base {};
```

- In class definition after class name and a colon put the list of base classes, if any
 - Sequence is important -> sequence of initialization if multiple base classes

```
class Base {};  
struct MixIn {};  
struct MultipleBases : public Base, private MixIn {};
```

- With interface inheritance, base class must be public
 - Private inheritance is possible, but only useful for mix-in classes that provide friend function
- Private inheritance can be used for some mix-in base classes that only add friend functions, like `boost/operators.hpp` helper classes
 - Most often, private base classes (with members) are wrong design!

- **Base constructors can be explicitly called in the member initializer list**
 - If a constructor of a base is omitted its default constructor is called
- **You should put base class constructor class before the initialization of members**
 - The compiler enforces this rule, even though you can put the list of initializers in wrong order

```
class DerivedWithCtor : public Base1, public Base2 {
    int mvar;
public:
    DerivedWithCtor(int i, int j) :
        Base1{i}, Base2{}, mvar{j} {}
};
```

```
struct Base1 {
    explicit Base1(int value) {
        std::cout << "Base1 with argument " << value << "\n";
    }
};

struct Base2 {
    Base2() { std::cout << "Base2\n"; }
};

class DerivedWithCtor : public Base1, public Base2 {
    int mvar;
public:
    DerivedWithCtor(int i, int j)
        : mvar{j}, Base2{}, Base1{mvar} {}
};

int main() {
    DerivedWithCtor dwc{1, 2};
}
```

- **C++' default mechanisms support value classes with copying/moving and deterministic lifetime**
- **Operator and function overloading and templates allow polymorphic behavior at compile time**
 - This is often more efficient and avoids indirection at run-time
- **Dynamic polymorphism needs object references or (smart) pointers to work**
 - Syntax overhead
 - The base interface must be a good abstraction
 - Copying carries the danger of slicing (an object is only copied partially)
- **Implementing design patterns for run-time flexibility: i.e., Strategy, Composite, Decorator**
 - Client code uses abstract interface and gets parameterized/called with reference to concrete instance
- **But: if run-time flexibility is not required, templates can implement many patterns with compile-time flexibility as well**

- If a function is reimplemented in a derived class it shadows its counterpart in the base class
- However, if accessed through a declared base object, the shadowing function is ignored
- The following example prints: Hi, I'm Base

```
struct Base {  
    void sayHello() const {  
        std::cout << "Hi, I'm Base\n";  
    }  
};  
  
struct Derived : Base {  
    void sayHello() const {  
        std::cout << "Hi, I'm Derived\n";  
    }  
};
```

```
void greet(Base const & base) {  
    base.sayHello();  
}  
  
int main() {  
    Derived derived{};  
    greet(derived);  
}
```


- **Dynamic polymorphism requires base classes with virtual member functions**
 - virtual member functions are bound dynamically
- **The following example prints: Hi, I'm Derived**

```
struct Base {  
    virtual void sayHello() const {  
        std::cout << "Hi, I'm Base\n";  
    }  
};  
  
struct Derived : Base {  
    virtual void sayHello() const {  
        std::cout << "Hi, I'm Derived\n";  
    }  
};
```



```
void greet(Base const & base) {  
    base.sayHello();  
}  
  
int main() {  
    Derived derived{};  
    greet(derived);  
}
```

- **virtual** is inherited and can be omitted in the derived class
- It is possible to mark an overriding function with **override**
 - Similar to the Java annotation `@Override` the compiler will produce an error if the annotated function does not override a member function in a base class

```
struct Base {  
    virtual void sayHello() const {  
        std::cout << "Hi, I'm Base\n";  
    }  
};  
  
struct Derived : Base {  
    void sayHello() const override {  
        std::cout << "Hi, I'm Derived\n";  
    }  
};
```

```
void greet(Base const & base) {  
    base.sayHello();  
}  
  
int main() {  
    Derived derived{};  
    greet(derived);  
}
```

- To override a virtual function in the base class the signature must be the same
- Constness of the member function belongs to the signature

```
struct Base {  
    virtual void sayHello() const {  
        std::cout << "Hi, I'm Base\n";  
    }  
};  
  
struct Derived : Base {  
    void sayHello() override {    
        std::cout << "Hi, I'm Derived\n";  
    }  
};  
  
struct OtherDerived : Base {    
    void sayHello(std::string name) const override {  
        std::cout << "Hi " << name << ", I'm OtherDerived\n";  
    }  
};
```

- **Value Object**

- Class type determines function, regardless of virtual

```
struct Base {  
    virtual void sayHello() const;  
};  
  
struct Derived : Base {  
    void sayHello() const;  
};  
  
void greet(Base base) {  
    //always calls Base::sayHello  
    base.sayHello();  
}
```

- **Reference**

- Virtual member of derived class called through base class reference

```
struct Base {  
    virtual void sayHello() const;  
};  
  
struct Derived : Base {  
    void sayHello() const;  
};  
  
void greet(Base const & base) {  
    //calls sayHello() of the actual type  
    base.sayHello();  
}
```

● Smart Pointer

- Virtual member of derived class called through smart pointer to base class

```
struct Base {
    virtual void sayHello() const;
};

struct Derived : Base {
    void sayHello() const;
};

void greet(std::unique_ptr<Base> base) {
    //calls sayHello() of the actual type
    base->sayHello();
}
```

● Dumb Pointer

- Virtual member of derived class called through base class pointer

```
struct Base {
    virtual void sayHello() const;
};

struct Derived : Base {
    void sayHello() const;
};

void greet(Base const * base) {
    //calls sayHello() of the actual type
    base->sayHello();
}
```

```
struct Animal {
    void makeSound() {cout << "---\n";}
    virtual void move() {cout << "---\n";}
    Animal() {cout << "animal born\n";}
    ~Animal() {cout << "animal died\n";}
};

struct Bird : Animal {
    virtual void makeSound() {cout << "chirp\n";}
    void move() {cout << "fly\n";}
    Bird() {cout << "bird hatched\n";}
    ~Bird() {cout << "bird crashed\n";}
};

struct Hummingbird : Bird {
    void makeSound() {cout << "peep\n";}
    virtual void move() {cout << "hum\n";}
    Hummingbird() {cout << "hummingbird hatched\n";}
    ~Hummingbird() {cout << "hummingbird died\n";}
};
```

```
int main() {
    cout << "(a)-----\n";
    Hummingbird hummingbird;
    Bird bird = hummingbird;
    Animal & animal = hummingbird;
    cout << "(b)-----\n";
    hummingbird.makeSound();
    bird.makeSound();
    animal.makeSound();
    cout << "(c)-----\n";
    hummingbird.move();
    bird.move();
    animal.move();
    cout << "(d)-----\n";
}
```

- What is the output?
- What is bad with this code's design?

- There are no interfaces in C++
- A pure virtual member function makes a class abstract
- To mark a virtual member function as pure virtual it has zero assigned after its signature

- = 0

- No implementation needs to be provided for that function

```
struct AbstractBase {  
    virtual void doitnow() = 0;  
};
```

- Abstract classes cannot be instantiated (like in Java)

```
AbstractBase create() {  
    return AbstractBase{};  
}
```

- **Classes with virtual members require a virtual Destructor**

- Otherwise when allocated on the heap with `std::make_unique<Derived>` and assigned to a `std::unique_ptr<Base>` only the destructor of Base is called

```
struct Fuel {  
    virtual void burn() = 0;  
    ~Fuel() { std::cout << "put into trash\n"; }  
};  
  
struct Plutonium : Fuel {  
    void burn() { std::cout << "split core\n"; }  
    ~Plutonium() { std::cout << "store many years\n"; }  
};  
  
int main() {  
    std::unique_ptr<Fuel> surprise = std::make_unique<Plutonium>();  
}
```

Output:
put into trash

- **Classes with virtual members require a virtual Destructor**

- Otherwise when allocated on the heap with `std::make_unique<Derived>` and assigned to a `std::unique_ptr<Base>` only the destructor of Base is called

```
struct Fuel {  
    virtual void burn() = 0;  
    virtual ~Fuel() { std::cout << "put into trash\n"; }  
};  
  
struct Plutonium : Fuel {  
    void burn() { std::cout << "split core\n"; }  
    ~Plutonium() { std::cout << "store many years\n"; }  
};  
  
int main() {  
    std::unique_ptr<Fuel> surprise = std::make_unique<Plutonium>();  
}
```

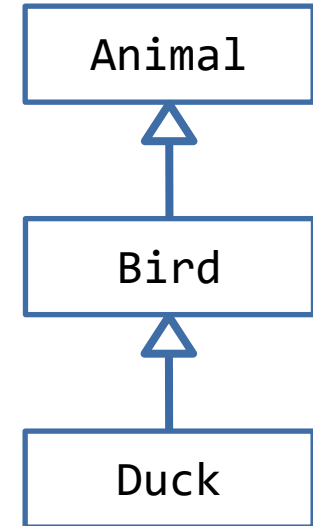
Output:
store many years
put into trash

- `std::shared_ptr` memorize the actual type and know which destructor to call

```
struct Fuel {  
    virtual void burn() = 0;  
    ~Fuel() { std::cout << "put into trash\n"; }  
};  
  
struct Plutonium : Fuel {  
    void burn() { std::cout << "split core\n"; }  
    ~Plutonium() { std::cout << "store many years\n"; }  
};  
  
int main() {  
    std::shared_ptr<Fuel> surprise = std::make_shared<Plutonium>();  
}
```

Output:
store many years
put into trash

- **Inheritance introduces a very strong coupling between subclasses and their base class**
 - You can hardly change the base class
- **API of base class must fit for all subclasses**
 - Very hard to get right
- **Conceptual hierarchies are often used as examples but are usually very bad software design, e.g., animal->bird->duck**
- **Only one standard library part (the oldest) uses inheritance with dynamic polymorphism: iostreams**



- **Assigning or passing by value a derived class value to a base class variable/parameter incurs object slicing**
 - Only base class member variables are transferred

```
struct Base {  
    int member{};  
    explicit Base(int initial) :  
        member{initial}{}  
    virtual ~Base() = default;  
    virtual void modify() { member++; }  
    void print(std::ostream & out) const;  
};  
  
struct Derived : Base {  
    using Base::Base;  
    void modify() {  
        member += 2;  
    }  
};
```

```
void modifyAndPrint(Base base) {  
    base.modify();  
    base.print(std::cout);  
}  
  
int main() {  
    Derived derived{25};  
    modifyAndPrint(derived);  
}
```

Output:
26

- **Member functions in derived classes hide base class member with the same name, even if different parameters are used**
 - Can be problematic, esp. with const/non-const
- **Example: `Derived::modify(int)` hides `Base::modify()`**

```
struct Base {  
    int member{};  
    explicit Base(int initial);  
    virtual ~Base() = default;  
    virtual void modify();  
};
```

hides

```
struct Derived : Base {  
    using Base::Base;  
    void modify(int value) {  
        member += value;  
    }  
};
```

```
int main() {  
    Derived derived{25};  
    derived.modify();  
    modifyAndPrint(derived);  
}
```

- By "using" the base class' member the hidden name(s) become visible

```
using Base::modify;
```

- This enables a call to `derived.modify()`

```
struct Base {  
    int member{};  
    explicit Base(int initial);  
    virtual ~Base() = default;  
    virtual void modify();  
};
```

```
struct Derived : Base {  
    using Base::Base;  
    using Base::modify;  
    void modify(int value) {  
        member += value;  
    }  
};
```

```
int main() {  
    Derived derived{25};  
    derived.modify();  
    modifyAndPrint(derived);  
}
```

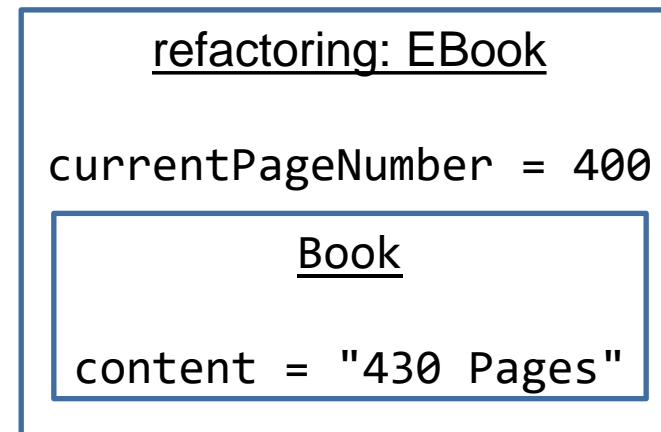
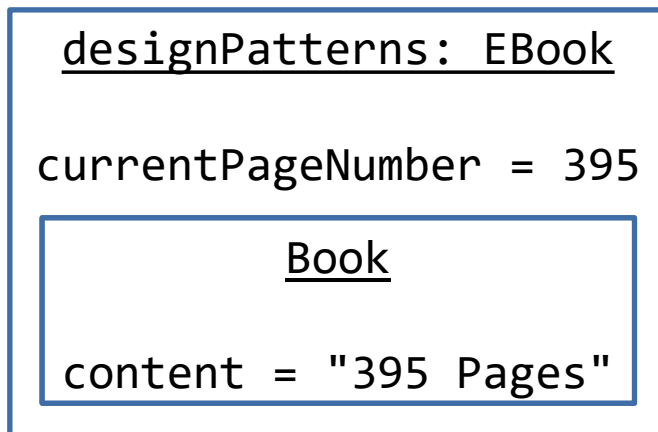
- Assignment cannot be implemented properly for virtual inheritance structures

```
struct Book {  
    explicit Book(std::vector<Page> pages) :  
        pages{pages}{}  
    virtual Page currentPage() const = 0;  
protected:  
    std::vector<Page> pages;  
};  
  
struct EBook : Book {  
    using Book::Book;  
    void openPage(size_t pageNumber);  
    Page currentPage() const;  
private:  
    size_t currentPageNumber{1};  
};
```

```
void readBook(Book book);  
  
int main() {  
    EBook designPatterns{"..."};  
    readBook(designPatterns);  
  
    EBook refactoring{"..."};  
    Book & some = designPatterns;  
    some = refactoring;  
}
```

- The assignment to the reference of the base class overwrites the Base part of the derived object

```
EBook designPatterns{writeEbook(395)};  
EBook refactoring{writeEbook(430)};  
refactoring.openPage(400);  
Book & some = refactoring;  
some = designPatterns;  
readPage(some.currentPage());
```



- You can declare the copy-operations as deleted

```
struct Book {  
    //...  
    Book & operator=(Book const & other) = delete;  
    Book(Book const & other) = delete;  
};  
  
struct EBook : Book {  
    //...  
    EBook(EBook const & other) :  
        Book{pages},  
        currentPageNumber{other.currentPageNumber}{}  
    EBook & operator=(EBook const & other) {  
        pages = other.pages;  
        currentPageNumber = other.currentPageNumber;  
        return *this;  
    }  
};
```

```
void readBook(Book book);  
  
int main() {  
    EBook designPatterns{"..."};  
    readBook(designPatterns);  
  
    EBook refactoring{"..."};  
    Book & some = designPatterns;  
    some = refactoring;  
    EBook copy = designPatterns;  
    copy = refactoring;  
}
```

- **You should only apply inheritance and virtual member functions if you know what you do**
- **Do not (like the IDE) create classes with virtual members by default**
- **If you design base classes with polymorphic behavior, understand the common abstraction that they represent**
 - Do not provide too many members or too few
 - Extract from existing class(es) the base after you see the commonality arise

- **Follow the Liskov Substitution Principle**

- Base class states must be valid for subclasses
- Do not break invariants of the base class
- Invariant signature: Member functions in subclasses must accept the same argument types as the base class (C++)
- Covariant return type: Return values must be inside the base class member function's range
- Don't change semantics unexpectedly

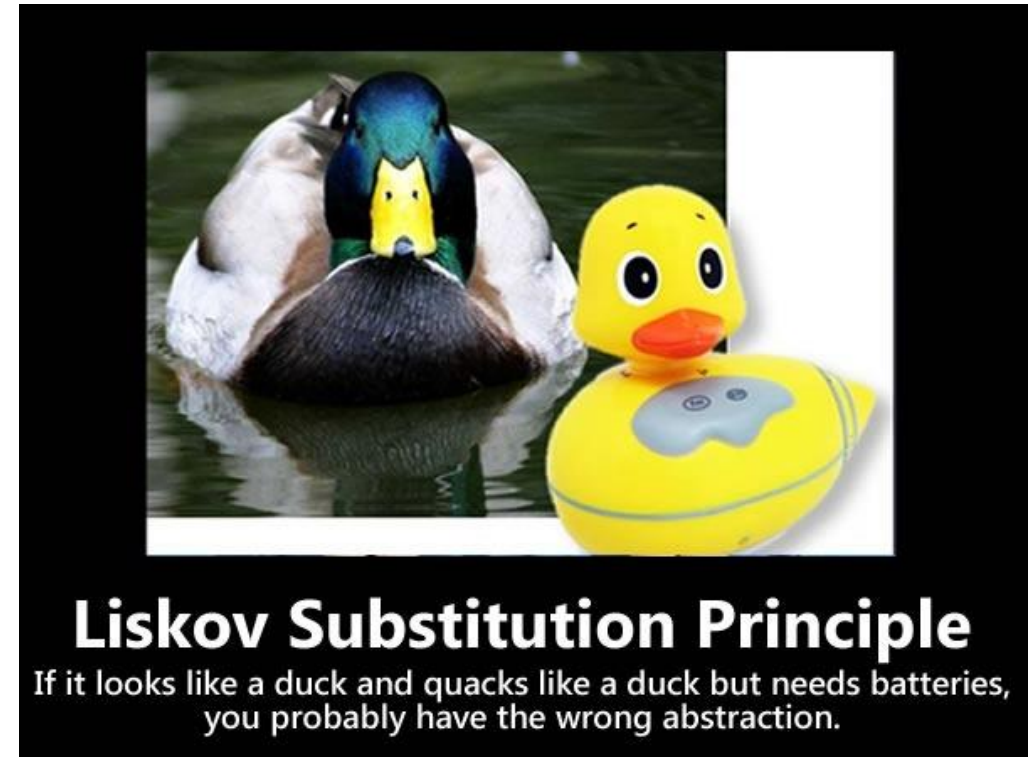


Image Source: http://www.globalnerdy.com/wordpress/wp-content/uploads/2009/07/liskov_substitution_principle.jpg

- **Three use cases:**
 - Inherit features from empty mix-in classes
 - Adapt features of a base class with a data-less subclass
 - Dynamic polymorphism
- **Beware of unwanted member hiding**
- **Avoid object slicing**
- **Mark Destructors virtual if you have any other virtual member function**